游易-程序主题分享合辑

+ 收藏专题

知识管理部 等 2022.06.07 15:37 6219 321 184个资源

汇总从101至今的游易征稿文章合辑。

浅谈内存优化

吴泽祥 2015.08.17 09:50 1515 10 4 查看原文

> 探讨一下内存优化的一些思路和经验，主要分享一些通用的内存池设计方案，笔者实现这些方案的一些经验，在多线程场景下可能遇到的问题以及相应的解决方案。

服务器性能优化涉及不同的内容，包括CPU、IO、网络，内存等等相关的方面。笔者前期做过一些优化的工作，想在这篇文章里面探讨一下内存优化的话题，分享一下自己的看法和经验。一家之言，仅作抛砖引玉之用。

1 | 适用场景

需要做内存优化的场景往往是，服务器有一系列需要管理的对象，这些对象的生成和销毁需要频繁地申请或释放内存。为了提高内存使用效率，减小内存碎片（包括内部碎片和外部碎片），此时，往往会通过设计一个内存池来达到目标。

2 | 内存池类型

一般而言，内存池有如下解决方案。

1. 固定大小的内存链表方案

每个内存块的大小固定，所有内存块串成一个单链表。这样的模型最为简单，实现起来也最容易。 这种内存池的关键参数是每个内存块的大小block_size确定，笔者之前做多线程IO优化的时候，实现了一个这样的内存池，当时候采取的方式去外服采样数据，分析数据之后确定这个block_size。另外，为了提高内存块插入和删除的效率，往往内存链表需要同时维护一个head和一个tail指针，分配内存时候tail指针+1，回收内存块的时候插入到head前面。此外，开始尽可能要申请足够一个运行周期的（比如游戏服务器一般是一周）内存块，避免重复多次申请。但由于服务器运行环境的复杂性，还是要支持链表的动态扩展，防止在内存池耗尽的情况下发生内存错误。方案示例图如下:

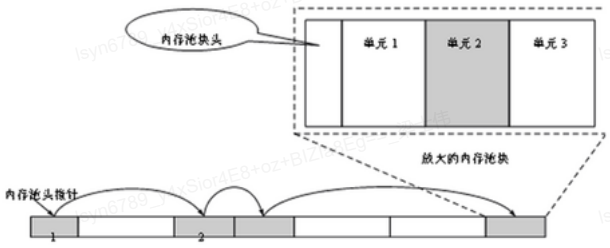

图1 固定大小内存链表

笔者这里给出前文所述的内存池的接口设计方案。如下:

```
        public:
                static SimpleMemPool * get_instance();

                struct mem_block * get_mem_block();
                void free_mem_block(struct mem_block * ptr);

                int get_len();

        private:
                SimpleMemPool(unsigned int);
                ~SimpleMemPool();

                size_t len;

                struct mem_block * head;
                struct mem_block * trace;

                static SimpleMemPool * instance;
};
```

图2 固定大小内存池接口设计

在get_mem_block接口中，如果原来的内存链表不够用了，要注意支持扩展。

2. 基于伙伴算法的内存池方案

伙伴算法是Linux内核采用的内存管理方案。笔者之前在km上有篇文章是分析其原理和简单实现的。简单来说，伙伴算法就是申请了一块足够大的内存，用一颗完全二叉树来管理这块内存，做到快速申请和释放，并且具有外部碎片小，使用率高的优点。伙伴算法简洁巧妙，在内核中的使用足以证明其优秀性，笔者也曾自己做了一个简单的实现，用在AOI优化中。其比较大的一个缺点可能在于较大的内部碎片。应用程序没有操作系统那么高的权限，做不到非常精细的管理，所以笔者见到的基于伙伴算法的应用层内存池倒并不太多。方案示例图如下：

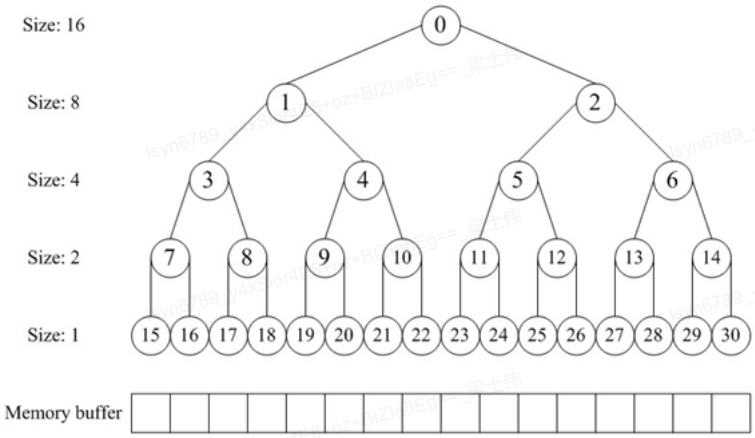

图3 基于伙伴算法的内存池

同样，这里给出笔者之前的一个基于伙伴算法的内存池接口设计示例：

```
class XY3Buddy {
        public:
                //pool_size:树根的大小*min_block_size
                //min_block_size:叶子节点每个节点的大小,单位为char
                XY3Buddy(unsigned int pool_size, unsigned int min_block_size);
                ~XY3Buddy();

                //一定会返回可用内存，优先向XY3Buddy，备选malloc
                void *AllocMemory(unsigned int mem_size, int &offset);
                void FreeMemory(int offset);

                int GetSize(int offset);

                int tree_size();

        private:
                //树管理结构
                int _tree_size;
                unsigned int *_longest;
                //内存
                void *_buffer;
                int _buffer_size;
                int _min_block_size;

                //私有成员函数
                //只会向XY3Buddy要内存，一定不会自己调用malloc
                void *__AllocMemory(unsigned int mem_size, int &offset);
};
```

图4 伙伴算法内存池接口设计

大小，对于需要提供给整个系统使用的内存池，固定大小的内存池并不能满足要求，可能会有非常大的内部碎片，也可能无法满足大对象的内存需求。在固定大小内存链表的基础上，按照不同对象大小（8字节，16字节，32，64，128，256，512，1K。。。64K），构造不同的固定内存分配器，这些分配器构成一个十字链表。分配内存时根据内存大小查表，决定到底由哪个分配器负责。内存块有一个header指针，分配后要在头部的 header 处标记分配器，以便释放的时候正确归还。如果大于64K，则直接用系统的 malloc作为分配。这个内存池的分配和释放都是接近O(1)的，但是会有一定的内存浪费（head指针，标记等所占内存）。 除了STL，memcached 的 slab 内存管理器，笔者之前所在项目所使用的内存池管理器，都是用的这个模型。这个模型有一个特点，申请到的内存一般不会主动释放，因此进程所使用的内存会随着运行时间不断膨胀。对于笔者熟悉的游戏服务器来说，服务器进程一般是外服上最重要的运行进程，而且一般定时重启，所以这个问题倒并不太大。方案示例图如下：

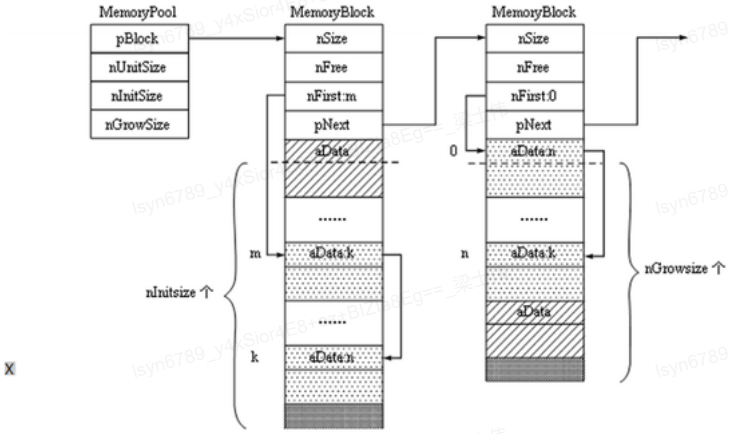

图5 free_list内存池

free_list内存池在以lua为主要开发语言的内存池中普遍使用，这里不再给出具体的使用例子。

4. 混合内存池方案

对于7*24运行的游戏服务器，往往有非常复杂的业务场景。此时单一的内存池方案是不足以解决所有问题的，需要在对业务场景归类分析的基础上，采用多种内存池方案组合的方式，达到最佳的性能。比如对于使用lua作为业务开发的项目。虚拟机支持自定制的内存分配器，这个时候可能free_list方案是提供给虚拟机的一个比较好的选择，运用这个方案的关键是要确定free_list里面每个固定list管理的内存块（block_size）的大小，这需要针对语言的特性做一些定制，主要是针对语言提供的不同数据类型的大小来确定block_size（需要研究下脚本的源代码）。但是在引擎层，针对一些特定的使用，比如文件IO的对象所使用的内存，则不妨使用固定大小的链表方案或是采用伙伴算法方案，既减少内存浪费，也提高了性能。

3 | 多线程相关

内存优化往往是其他优化的一个附属品。这里的其他优化指的一般是CPU，IO相关的优化，这些又往往涉及到了并发。一个常见的使用场景是，A线程从内存池M申请了一块内存ma，把这块内存内存对应的指针pm丢给线程B，B做对应的工作。B完成之后，针对B需不需要通知A结果，对于如何回收内存，有两种方案。一是不需要，那么这个时候可以由B释放，也可以由B通知A释放，前一种情况涉及到对内存池的并发访问，需要对内存池加锁，后一种情况需要对结果队列加锁，两种情况都有一次加锁的操作，所以两种方式都可以，一般来说会采用B释放的方式，更加简单。另外一个方案针对B需要通知A结果的情况，这个时候一般采用由A来释放的方式。A收到B的结果之后，同时释放pm，因为内存池只有A操作，不存在并发，这样的方案可以减少一次内存池的加锁，提高性能（通知结果的队列那次加锁是免不了的）。

4 | 结语

上面讨论了一些经典的内存池解决方案。对于现代操作系统来说，其实系统自带的内存池管理器已经工作得很好了，freebsd上的jemalloc，google的tcmalloc都已经被无数实践证明了其优越的性能。重头造轮子做一个内存管理器其实意义并不大。笔者自己的一些实践也证明了，这样做带来的提升并不太大，而且为了完成内存池还有可能要花费很多设计和调试的精力，尤其是在多线程场景下，性价比不高。所以并不太建议有这样想法的同学自己从头做一个。当然，针对非常确定的使用场景，使用一些简单的内存池来做优化，还是有价值的，这个往往就要具体问题具体分析了。

请输入评论内容

还可以输入 500 个字

(可添加1个视频+5张图片) 匿名 评 论

最热 最新

Wade(吴建江)

4楼 666

2017-10-17 14:12 回复 0

匿名

3楼 赞赞赞

2015-12-18 11:21 回复 0

匿名

2楼 打个酱油。

2015-09-02 12:21 回复 0

匿名

1楼 赞

2015-08-28 10:40 回复 0

加载完毕,没有更多了